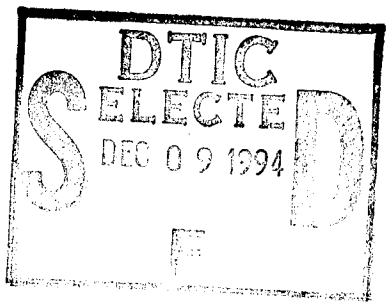
UNCLASSIFIED



AFIT/EN/TR/94-09

Air Force Institute of Technology

An Ada Binding for ODMG-93

Stephen R. Lindsay, Capt, USAF
Mark A. Roth, Lt Col, USAF

10 Nov 94

Approved for public release; distribution unlimited

An Ada Binding for ODMG-93

Stephen R. Lindsay
Mark A. Roth*
Air Force Institute of Technology
2950 P St, Bldg 642
Wright-Patterson AFB, Ohio 45433-7765
(513) 255-2024 FAX: (513) 476-4055
{slindsay|mroth}@hawkeye.afit.af.mil

Abstract

An Ada 9X binding to the proposed object database standard, ODMG-93, is presented. The major decisions necessary for such an effort are described. The approach allows Ada programmers to use a single language to access object-oriented database functionality regardless of vendor. Briefly mentioned is our successful feasibility test using an Ada 83 compiler with Ada package implementations for ObjectStore and ITASCA.

1 Introduction

This paper identifies the major decisions involved in creating an Ada language binding for the Object Definition Language (ODL), Object Manipulation Language (OML), and Object Query Language (OQL) proposed in the ODMG-93 object database standard [1:11–81]. As with the other ODMG-93 language bindings, the primary goal is for Ada programmers to feel they are using a single language to access the underlying ODBMS functionality.

Similar to the steps in creating a C++ ODBMS application [1:86], Figure 1 illustrates the process for creating an application in Ada. Object declarations in the form of package specifications are scanned by the ODL preprocessor, which translates them to the corresponding schema and database files in addition to generating any required auxiliary packages. An Ada OML preprocessor is not required, as all ODMG-93 constructs manipulating objects may be defined in Ada 9X with respect to a root ancestor object (Section 2). Likewise, an Ada OQL preprocessor is not required due to the loosely coupled approach for implementing queries (Section 5). The resulting source code along with the Ada binding is compiled and then linked with the ODBMS to produce an application.

The object database standard integrates well with Ada's packaging approach to encapsulation and information hiding; a single package specification independent of any vendor's ODBMS can be used to provide access to a category of ODMG-93 constructs using the syntax set forth in the standard. The corresponding vendor-specific package bodies can implement them for a particular

*Current address for Mark Roth: USSTRATCOM/J673, 901 SAC Blvd, STE 2B10, Offutt AFB, NE 68113-6600, (402) 294-4616, FAX: (402) 294-1020, rothm@j67.stratcom.af.mil

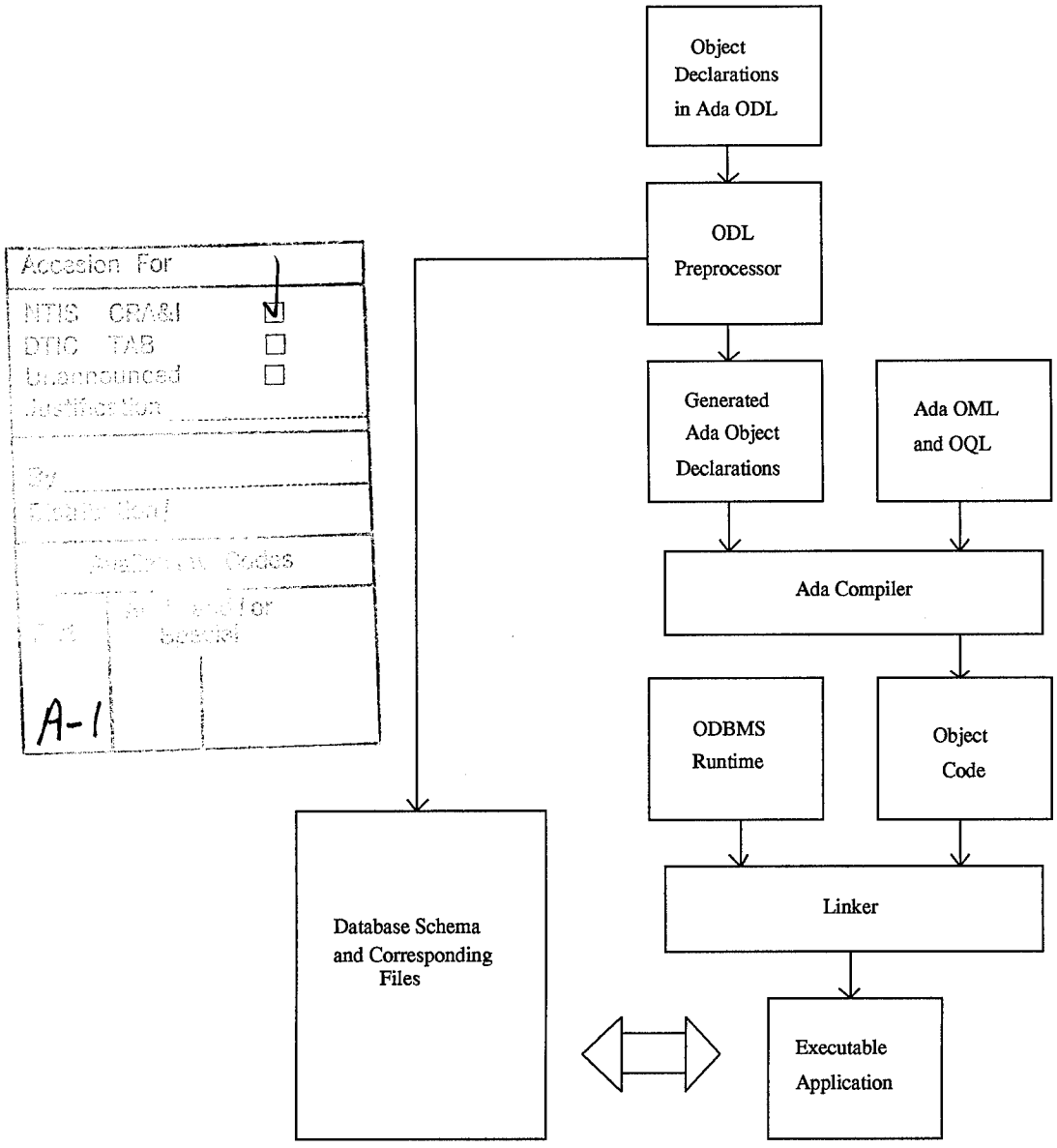


Figure 1: Creating an Executable Ada ODBMS Application

system. Ada's use clauses can then be incorporated to make the constructs appear as a natural extension to the language.

Ada 83 is the language commonly referred to simply as "Ada," but because of its inability to model inheritance it cannot be used with the standard unless a convention for modeling inheritance is decided upon. Ada 9X, the object-oriented version of Ada 83 currently in the final standardization process, is the only Ada language that can be used "as is" with ODMG-93. As the new compilers will soon become available this paper concentrates on Ada 9X, and unless explicitly stated, "Ada" will reference Ada 9X instead of Ada 83.

In the rest of this paper we discuss the binding approach we used for Ada 9X utilizing the new inheritance capabilities of the language. We then discuss and provide examples of the binding for the ODL, OML, and OQL in ODMG-93. Finally, we discuss our feasibility test using Ada 83 and package bodies for ObjectStore and ITASCA. The complete binding is presented in Appendix A.

2 Language Binding Approach

The most important decision for the Ada ODMG language binding is the overall approach to object definition and manipulation, as this decision then dictates the definition of all remaining ODL and OML constructs. The approach chosen for Ada is termed *inheritance-based.*

This approach is similar to the "Ref-based" approach for the ODMG-93 C++ binding [1:84–85], which refers to instances of persistence-capable database classes using the template class `Ref`, or a reference to the instance. In the Ref-based approach, the ODL preprocessor automatically defines the class `Ref<X>` for every database class `X` specified by the user. The template class allows objects to be accessed in a manner similar to C++ pointer types, with additional facilities for guaranteeing integrity in pointers to persistent objects [1:99]. Furthermore, the class `Pobject` is defined as a superclass of all persistence-capable objects, allowing the parameters to ODMG contructs to be defined as `Ref<Pobject>` [1:108]. The ability of Ada 9X to model inheritance allows it to use an approach quite similar to the C++ Ref-based approach.

Figure 2 illustrates one possible declaration for package `Persistent`, from which all persistence-capable database classes may be derived. A new persistent object is created and placed in a database "near" (implementation dependent) a given clustering object, if provided. Like the C++ template class `Ref<>`, additional mechanisms for guaranteeing integrity can be incorporated in this package.

A persistence-capable class `X` may now be defined using a modified version of ROMAN-9X notation [2:390], one of several techniques for modeling inheritance in Ada 9X, as shown in Figure 3. ROMAN-9X defines the object record using keyword `private`, preventing the user from accessing these attributes directly (as shown in Figure 2). Here, the record definition is moved to the public section of the package, eliminating the need to define accessor routines for each attribute. New attributes may be added within the record, and methods may be introduced as procedures and functions. Figure 4 shows the definition of an example ODMG-93 operation using the inheritence-based approach.

3 Ada ODL

Having selected the approach for designing the interface, the Ada language binding for the remaining ODMG-93 constructs follows naturally. The database schema is defined as a set of packages, each

```
package Persistent is

  type Object    is private;
  type Reference is access all Object'Class;

  function Create(A_Database : in Database.Object;
                  Clustering : in Reference := null)
    return Object;
  procedure Delete(An_Object : in Object);
  procedure Mark_Modified(An_Object : in out Object);

private

  String_Max : constant := 300;
  type Object is
    record
      Name     : String(1..String_Max);
      Modified : Boolean;
    end record;

end Persistent;
```

Figure 2: Superclass For Ada Persistence-capable Classes

```
with Persistent;
package X is

  type Object is new Persistent.Object with
    record
      Attribute_One : Attribute_One_Type;
      Attribute_Two : Attribute_Two_Type;
    end record;

  type Reference is access all Object'Class;

  ...

end X;
```

Figure 3: Example Ada Persistence-capable Class

```
procedure Name_Object(An_Object : in out Persistent.Object;
                      Name      : in     String);
```

Figure 4: Example Inheritence-based Ada Operation

```
with Person, Student, Set;
package Faculty is

  package Student_Set is new Set(Student.Object);

  type Object is new Person.Object with
    record
      ...
      Advisees : Student_Set.Object inverse Student.Advisor;
    end record;
  ...
end Faculty;
```

Figure 5: Example Relationship Definition

representing a unique object class. As shown in Figure 3, attributes are defined for an object class in Ada by extending the attributes of the record defined in the superclass package.

Relationships between objects may be defined using the collection generic `Set` (discussed in Section 4.2) and keyword `inverse`, specifying inverse traversal paths. Because the keyword `inverse` is not part of standard Ada, it will need to be removed by the preprocessor before compilation. An Ada object class may then declare a relationship using the syntax shown in Figure 5.

Consistent with the definition of any Ada abstract data type, operations are defined by the procedures and functions in the package representing an object class.

4 Ada OML

This section discusses binding ODMG-93's OML to Ada. Every effort must be aimed toward adherence to the principle that manipulating persistent objects is syntactically equivalent to manipulating transient objects.

4.1 Object Creation, Deletion, and Modification

The object creation and deletion operations are best defined in package `Persistent` and refined in the persistent-capable packages. Consistent with the other ODMG-93 language bindings, Ada

```
with Persistent, Database, Person;
package Faculty is
  ...
function
  Create(Database   : in Database.Object;
         Clustering : in Persistent.Reference := null;
         Name       : in Name_String          := (others => ' ');
         SSAN       : in Integer              := 0;
         Birthday   : in Birthday_String      := (others => ' ');
         Dept       : in Department_Type      := NO_DEPT;
         Salary     : in Float                := 0.0)
    return Object is
  Instance : Object;
begin
  Instance := Person.Create(Database, Clustering, Name,
    SSAN, Birthday);
  Instance.Dept := Dept;
  Instance.Salary := Salary;
  return Instance;
end Create;
```

Figure 6: Example Object Creation

programmers must have the ability to manipulate both persistent and transient instances of objects. Overloading the `Create` and `Delete` operations can accomplish this.

To create persistent instances, the user must produce a `Create` function with a parameter of type `Database.Object`; to incorporate clustering, an additional parameter of type `Persistent.Reference` may be included. The same applies to the `Delete` procedure. Attribute initial values are assigned as default values in the function stub. Figure 6 illustrates this concept. Similar to creating an instance of an inherited class in C++, the ancestor instances are created first. When the instance of root class `Persistent` is created, the persistent object is placed in the corresponding database near the given clustering object, if applicable.

If the user also wishes to manipulate transient instances, he or she is responsible for defining one or more additional `Create` and `Delete` routines without including the database or clustering parameters. These routines will simply be standard Ada functions and procedures based on the user's needs. Consistent with the C++ binding, all other operations for manipulating persistent objects can also be used to manipulate transient objects, with the exception of queries and transactions.

An object can be modified by manipulating the attribute and relationship properties defining its state.

```
with Persistent, Collection;
package Relationships is

  package Persistent_Set is new Set(Persistent.Object);

  procedure Relate_One_To_Many_Create
    (Relationship : in     String;
     An_Object    : in out Persistent.Object;
     Related_Set  : in     Persistent_Set.Object);

  procedure Relate_One_To_One_Create
    (Relationship   : in     String;
     An_Object      : in out Persistent.Object;
     Related_Object : in out Persistent.Object);

  ...
```

Figure 7: Example Relationship Creation Operations

4.1.1 Attributes

Attributes may be manipulated by the programmer directly using Ada's standard notation for manipulating record fields. If the record definition is moved to the private section of the package, accessor procedures and functions for each attribute will be required. However, these operations may be added by the ODL preprocessor rather than the programmer. No accessor procedures or functions should be allowed on relationship attributes, as these are best manipulated with the operations in package `Relationship`.

4.1.2 Relationships

Package `Relationship` can be used to define the ODMG-93 relationship operations. Figure 7 illustrates the creation operations for one-to-one and one-to-many relationships. One-to-one relationships must be implemented as pointers to other objects, rather than a copy of the object, to preserve data integrity. Because collections are defined as pointers (in Section 4.2), an additional level of dereferencing is not required for one-to-many and many-to-many relationships.

4.2 Collections

Similar to the templates used to implement collections in the C++ binding, Ada generic packages should be used to implement collections. In the Ada binding, `Collection` simply provides a way to encapsulate the properties and operations common to all collection subclasses; it is not an instantiable abstract data type. The proposed designs of package `Collection` and each of its subclasses appear in Appendix A. Each subclass defines its own object type as `new Collection.Object` to

```
package Faculty_List is new List(Faculty.Object);

Chairman,
Vice_Chairman      : Faculty.Object;
Math_Faculty_List : Faculty_List.Object :=
  Faculty_List.Create;

Faculty_List.Insert_First_Element(Chairman,
  Math_Faculty_List);
Faculty_List.Insert_Element_After(Vice_Chairman, 1
  Math_Faculty_List);
```

Figure 8: Example `List` Instantiation and Manipulation

allow visibility to the collection properties and operations. The array collection is renamed `Array_Type` to avoid illegal use of Ada keyword `array`. Figure 8 illustrates an example instantiation and manipulation of a `List` object in Ada.

The type representation for collections is chosen here to be a subtype of `System.Address`. The underlying ODBMS is therefore responsible for converting its collection type representation, thus allowing collections to be manipulated as pointers in the Ada binding. Iterators are defined in a similar manner.

4.3 Transactions

Package `Transaction` can be defined to encapsulate all transaction operations. Because a transaction can be represented in several ways in an ODBMS, type `Transaction.Object` is best defined as a subtype of `System.Address`. The underlying implementation will then be responsible for converting its particular transaction representation type to the address type. Function `Begin`—returning a `Transaction.Object`—along with procedures `Commit_Txn`, `Abort_Txn`, `Checkpoint`, and `Abort_To_Top_Level`—requiring transaction parameters—may then be defined. The ODMG-93 `abort` operation should be renamed `Abort_Txn` since the former is an Ada keyword. For consistency, the commit operation be renamed `Commit_Txn`.

4.4 Database Operations

Package `Database` can be defined similar to the transaction package. Type `Database.Object` then represents the address of a database file. Function `Open` returns the address of the file referred to by the string parameter, and `Close` operates as expected. Operations `Name_Object` and `Lookup_Object` manipulating persistent object names in the corresponding database.

```
Faculty_Set is new Set(Faculty.Object);
Jones_Advisors : Faculty_Set.Object :=
  Faculty_Set.Select_Subcollection(
    "exists S in Advisees: S.Last_Name = 'Jones'");
```

Figure 9: Example Collection Query

```
Student_Set is new Set(Student.Object);
A_Student_Set : Student_Set.Object :=
  Database.Query(My_Database,
    "select S " &
    "from S in Students, F in Faculty " &
    "where Abs(S.SSAN - F.SSAN) <= 5 and " &
    "S in Relationship_One_To_Many_Traverse(F, 'Advisees')");
```

Figure 10: Example Database Query

5 Ada OQL

Consistent with the other ODMG-93 OQL language bindings, the loosely coupled approach is recommended in the initial Ada binding, with predicates introduced as strings that are parsed, optimized, and evaluated at runtime rather than at compile time. An ODBMS implementation of Ada OQL will therefore require a runtime parser to translate the predicate from Ada syntax to the syntax required by the underlying system.

5.1 Collection Queries

Collections can be filtered using the two operations defined for each collection generic package: `Select_Subcollection` and `Select_Element`. Each takes the predicate string as its sole parameter. For example, assuming extent `Faculties` contains all instances of object class `Faculty`, the query in Figure 9 returns all `Faculty` instances advising a student with a last name of "Jones."

5.2 Database Queries

More complex query operations defined over an entire database should also be defined. Figure 10 illustrates an example query selecting all students whose SSANs are within 5 points of their advisors.

```
subtype Object is System.Address;

procedure Name_Object(An_Object : in out Object;
                      Name      : in     String);
```

Figure 11: Example Address-based Operation

6 Producing and Implementing an Ada 83 Binding

We tested our Ada binding by writing vendor specific package bodies for the ObjectStore and ITASCA ODBMS products. We used Sun Ada 83 on a Sun Sparcstation 2 as the compiler. The issues we dealt with were Ada 83's inability to model inheritance and the current lack of an ODMG-93 standard interface for the two ODBMS products. Below we briefly discuss the Ada 83 issues and our implementation efforts.

6.1 A New Approach: Address-based

Ada 83 requires a different approach for defining the language binding: *address-based.* This approach uses pointers by defining the high-level database types such as `Object` and `Collection.Set` as a subtype of the platform-dependent `System.Address`. Figure 11 shows the redefinition of the ODMG-93 construct of Figure 4 using the address-based approach.

This technique allows the definition of address parameters to and from the various ODMG-93 constructs without regard to the data they reference. If, however, an address referencing an invalid or inappropriate data structure is passed to an ODMG operation, an exception is raised. This method removes the need to overload operations based on all the valid data types the user may specify; however, it does require the majority of object related error-checking to be performed at runtime rather than at compile time.

6.2 Additional Considerations

Attribute and method definition in Ada 83 packages required a more sophisticated ODL preprocessor than that for Ada 9X. A convention had to first be decided upon to communicate inheritance between packages to the preprocessor, which could then use its native constructs to make a corresponding representation.

Implementing an Ada 83 binding for an ODBMS is further complicated by the nonexistence of any commercial ODBMS implemented in Ada 83. As a result, an implementation must use a programming language to which Ada can link using its `pragma` commands. If this language does not have a mechanism for specifying inheritence, a simulation technique must be devised.

6.3 Prototype Implementation

The address-based approach mentioned in Section 6.1 was implemented using the C programming language interface libraries for the ITASCA and ObjectStore ODBMS products [4]. Defining the

high-level types as subtypes of `System.Address` facilitated implementing the binding; however, for the reasons stated in Section 6.2, the ObjectStore ODL preprocessor required a much greater degree of involvement than the ODL preprocessor for ITASCA.

The primary reason for this occurrence is the strong typing of ObjectStore's implementation language. The C language implementation accessing ObjectStore's functionality necessitated the creation of parallel data types before attribute values could be manipulated. This required the class of the object to be determined so that its address could be appropriately converted, all of which must be decided at compile time. ITASCA, on the other hand, has no requirement for declaring parallel data types. Although its functionality was accessed using the C interface, its ODBMS implementation language is Lisp, a weakly typed language. As a result, no parallel data types were required and attribute values could be manipulated using the object-oriented message passing techniques for Lisp and CLOS.

Additionally, the ODBMS was required to model inheritance for Ada 83. ITASCA's ability to functionally specify inheritance between classes allowed inherited attributes to be manipulated in exactly the same way as non-inherited attributes. Because C cannot model inheritance, a technique for its simulation was required. This further required the ODL preprocessor to resolve the simulation technique when inherited attributes were accessed.

6.4 Summary of Results

Operating under the assumed existence of ODL preprocessors for each ODBMS, we were able to produce a portable Ada database application. With the simple process of exchanging the vendor-specific package bodies, the Ada binding and the application could be recompiled and executed using both ObjectStore and ITASCA. The ObjectStore version could take advantage of the sophisticated performance-based memory-mapping and clustering techniques inherent in this ODBMS[1], although its implementation was more preprocessor-dependent than its counterpart. The ITASCA version exploited the language neutrality of its underlying system to produce an elegant and straightforward implementation.

The ability to produce portable Ada/ODBMS applications that can take advantage of the unique strengths of any system represents the fundamental motivation behind ODMG-93. Its evolution as well as the evolution of the resulting language bindings will give ODBMS programmers a powerful facility in the near future.

References

[1] ATWOOD, T., ET AL. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[2] CERNOSEK, G. ROMAN-9X: A technique for representing object models in Ada 9X notation. *Association for Computer Machinery* (1993), 385–406.

[3] HALLORAN, T. J. Performance measurement of three commercial object-oriented database management systems. Master's thesis, Air Force Institute of Technology (AETC), Dec. 1993.

[1] See [3] for a detailed comparison of ObjectStore and ITASCA performance.

[4] LINDSAY, S. R. Designing and implementing an Ada language binding specification for ODMG-93. Master's thesis, Air Force Institute of Technology (AETC), Dec. 1994.

A Proposed Ada Binding

This appendix contains our complete Ada binding for the proposed ODMG-93 standard, organized by Ada package.

A.1 Persistent

```
with Database;
package Persistent is
  type Object    is private;
  type Reference is access all Object'Class;

  function  Create(A_Database : in Database.Object;
                   Clustering : in Reference := null)
    return Object;
  procedure Delete(An_Object : Object);
  procedure Mark_Modified(An_Object : in Object);

private
  String_Max : constant := 300;
  type Object is
    record
      Name     : String(1 .. String_Max);
      Modified : Boolean;
    end record;
end Persistent;
```

A.2 Database

```
with System, Persistent, Collection;
package Database is
  subtype Object is System.Address;

  function Open_Database(Name : in String)
    return Object;
  procedure Close_Database(A_Database : in Object);
  procedure Name_Object(An_Object  : in out Persistent.Object;
                        Name       : in     String;
                        A_Database : in     Object);
  function Lookup_Object(Name       : in String;
                         A_Database : in Object)
    return Persistent.Object;
```

```
  package Persistent_Collection is new
    Collection(Persistent.Object);
  function Query(A_Database : in Object;
                 Predicate  : in String)
    return Persistent_Collection.Object;
  function Query(A_Database : in Object;
                 Predicate  : in String)
    return Persistent.Object;
end Database;
```

A.3 Transaction

```
with System;
package Transaction is
  subtype Object is System.Address;

  function Start return Object;
  procedure Commit_Txn(A_Transaction : in Object);
  procedure Abort_Txn(A_Transaction  : in Object);
  procedure Checkpoint(A_Transaction : in Object);
  procedure Abort_To_Top_Level;
end Transaction;
```

A.4 Relationships

```
with Persistent, Set;
package Relationships is
  package Persistent_Set is new Set(Persistent.Object);
  procedure Initialize;

  procedure Relate_One_To_One_Create
    (Relationship   : in     String;
     An_Object      : in out Persistent.Object;
     Related_Object : in out Persistent.Object);
  procedure Relate_One_To_One_Delete
    (Relationship   : in     String;
     An_Object      : in out Persistent.Object);
  function Relate_One_To_One_Traverse
    (Relationship   : in String;
     An_Object      : in Persistent.Object) return Persistent.Object;

  procedure Relate_One_To_Many_Create
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Related_Set      : in     Persistent_Set.Object);
```

```
  procedure Relate_One_To_Many_Delete
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object);
  procedure Relate_One_To_Many_Add_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Object_To_Add    : in out Persistent.Object);
  procedure Relate_One_To_Many_Remove_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Object_To_Remove : in out Persistent.Object);
  function Relate_One_To_Many_Traverse
    (Relationship     : in String;
     An_Object        : in Persistent.Object) return Persistent_Set.Object;

  procedure Relate_Many_To_Many_Delete
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object);
  procedure Relate_Many_To_Many_Add_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Object_To_Add    : in out Persistent.Object);
  procedure Relate_Many_To_Many_Remove_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Object_To_Remove : in out Persistent.Object);
  procedure Relate_Many_To_Many_Add_One_To_Many
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Set_To_Add       : in     Persistent_Set.Object);
  procedure Relate_Many_To_Many_Remove_One_To_Many
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object;
     Set_To_Remove    : in     Persistent_Set.Object);
  procedure Relate_Many_To_Many_Remove_All_From
    (Relationship     : in     String;
     An_Object        : in out Persistent.Object);
end Relationships;
```

A.5 Iterator

```
with System;
generic
  type Collected_Object is private;
package Iterator is
```

```
  subtype Object is System.Address;

  function  More(An_Iterator : in Object) return Boolean;
  procedure First(An_Iterator : in out Object;
                  An_Object   :    out Collected_Object);
  procedure Last(An_Iterator : in out Object;
                 An_Object   :    out Collected_Object);
  procedure Next(An_Iterator : in out Object;
                 An_Object   :    out Collected_Object);
  procedure Reset(An_Iterator : in out Object);
  procedure Delete(An_Iterator : in out Object);
end Iterator;
```

A.6 Collection

```
with System, Iterator;
generic
  type Collected_Object is private;
package Collection is
  Cardinality       : Integer;
  Empty             : Boolean;
  Ordered           : Boolean;
  Allows_Duplicates : Boolean;

  subtype Object          is System.Address;
  subtype Iterator_Object is System.Address;

  function  Create return Object;
  procedure Delete(A_Collection : in out Object);
  function  Copy(A_Collection : in Object) return Object;
  procedure Insert_Element(An_Object    : in     Collected_Object;
                           A_Collection : in out Object);
  procedure Remove_Element(An_Object    : in     Collected_Object;
                           A_Collection : in out Object);
  function  Select_Element(A_Collection : in Object;
                           Predicate    : in String)
    return Collected_Object;
  function  Select_Subcollection(A_Collection : in Object;
                                 Predicate    : in String)
    return  Object;

  function  Contains_Element(A_Collection : in Object;
                             An_Object    : in Collected_Object)
    return  Boolean;
```

```
  function  Create_Iterator(A_Collection : in Object)
    return Iterator_Object;
end Collection;
```

A.7 Set

```
with System, Collection;
generic
  type Collected_Object is private;
package Set is
  package Set_Collection is new Collection(Collected_Object);
  type Object is new Set_Collection.Object;

  function Create return Object;
  procedure Insert_Element(An_Object : in     Collected_Object;
                           A_Set     : in out Object);
  function Union(Set_One : in Object;
                 Set_Two : in Object) return Object;
  function Intersection(Set_One : in Object;
                        Set_Two : in Object) return Object;
  function Difference(Set_One : in Object;
                      Set_Two : in Object) return Object;
  function Is_Subset(Set_One : in Object;
                     Set_Two : in Object) return Boolean;
  function Is_Proper_Subset(Set_One : in Object;
                            Set_Two : in Object) return Boolean;

  function Is_Superset(Set_One : in Object;
                       Set_Two : in Object) return Boolean;

  function Is_Proper_Superset(Set_One : in Object;
                              Set_Two : in Object) return Boolean;
end Set;
```

A.8 Bag

```
with System, Collection;
generic
  type Collected_Object is private;
package Bag is
  package Bag_Collection is new Collection(Collected_Object);
  type Object is new Bag_Collection.Object;

  function  Create return Object;
  procedure Insert_Element(An_Object : in     Collected_Object;
                           A_Bag     : in out Object);
```

```
  procedure Remove_Element(An_Object : in     Collected_Object;
                           A_Bag     : in out Object);
  function  Select_Subcollection(A_Bag     : in Object;
                                 Predicate : in String)
    return  Object;
  function Union(Bag_One : in Object;
                 Bag_Two : in Object) return Object;
  function Intersection(Bag_One : in Object;
                        Bag_Two : in Object) return Object;
  function Difference(Bag_One : in Object;
                      Bag_Two : in Object) return Object;
end Bag;
```

A.9 List

```
with System, Collection;
generic
  type Collected_Object is private;
package List is
  Current_Position : Integer;
  package List_Collection is new Collection(Collected_Object);
  type Object is new List_Collection.Object;

  function  Create return Object;
  procedure Insert_Element(An_Object : in     Collected_Object;
                           A_List    : in out Object);
  function  Select_Subcollection(A_List    : in Object;
                                 Predicate : in String)
    return  Object;
  procedure Insert_Element_After(An_Object : in     Collected_Object;
                                 Position  : in     Integer;
                                 A_List    : in out Object);
  procedure Insert_Element_Before(An_Object : in     Collected_Object;
                                  Position  : in     Integer;
                                  A_List    : in out Object);
  procedure Insert_First_Element(An_Object : in     Collected_Object;
                                 A_List    : in out Object);
  procedure Insert_Last_Element(An_Object : in     Collected_Object;
                                A_List    : in out Object);
  procedure Remove_Element_At(Position : in     Integer;
                              A_List   : in out Object);
  procedure Remove_First_Element(A_List : in out Object);
  procedure Remove_Last_Element(A_List : in out Object);
  procedure Replace_Element_At(An_Object : in     Collected_Object;
                               Position  : in     Integer;
```

```
                                A_List    : in out Object);
  function  Retrieve_Element_At(Position : in Integer;
                                A_List   : in Object)
    return Collected_Object;
  function  Retrieve_First_Element(A_List : in Object)
    return Collected_Object;
  function  Retrieve_Last_Element(A_List : in Object)
    return Collected_Object;
end List;
```

A.10 Array_Type

```
with System, Collection;
generic
  type Collected_Object is private;
package Array_Type is
  package Array_Collection is new Collection(Collected_Object);
  type Object is new Array_Collection.Object;

  procedure Insert_Element_At(An_Object : in     Collected_Object;
                              Position  : in     Integer;
                              An_Array  : in out Object);

  procedure Remove_Element_At(Position : in     Integer;
                              An_Array : in out Object);

  procedure Replace_Element_At(An_Object : in     Collected_Object;
                               Position  : in     Integer;
                               An_Array  : in out Object);

  function  Retrieve_Element_At(Position : in Integer;
                                An_Array : in Object)
    return Collected_Object;

  procedure Resize(New_Size : in     Integer;
                   An_Array : in out Object);
end Array_Type;
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1994	Technical Report

4. TITLE AND SUBTITLE
An Ada Binding for ODMG-93

5. FUNDING NUMBERS

6. AUTHOR(S)
Stephen R. Lindsay, Mark A. Roth

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN/TR/94-09

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
WL/AAWA
Wright Laboratory
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

An Ada 9X binding to the proposed object database standard, ODMG-93, is presented. The major decisions necessary for such an effort are described. The approach allows Ada programmers to use a single language to access object-oriented database functionality regardless of vendor. Briefly mentioned is our successful feasibility test using an Ada 83 compiler with Ada package implementations for ObjectStore and ITASCA.

14. SUBJECT TERMS
Object-Oriented Databases, Ada, ODMG-93, ODMG

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL